Process Sleep and Wakeup on a Shared-memory Multiprocessor

Rob Pike
Dave Presotto
Ken Thompson
Gerard Holzmann

{rob|presotto|ken|gerard}@research.att.com
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The problem of enabling a 'sleeping' process on a shared-memory multiprocessor is a difficult one, especially if the process is to be awakened by an interrupt-time event. We present here the code for sleep and wakeup primitives that we use in our multiprocessor system. The code has been exercised by months of active use and by a verification system.

Our problem is to synchronise processes on a symmetric shared-memory multiprocessor. Processes suspend execution, or *sleep*, while awaiting an enabling event such as an I/O interrupt. When the event occurs, the process is issued a *wakeup* to resume its execution. During these events, other processes may be running and other interrupts occurring on other processors.

More specifically, we wish to implement subroutines called `sleep`, callable by a process to relinquish control of its current processor, and `wakeup`, callable by another process or an interrupt to resume the execution of a suspended process. The calling conventions of these subroutines will remain unspecified for the moment.

We assume the processors have an atomic test-and-set or equivalent operation but no other synchronisation method. Also, we assume interrupts can occur on any processor at any time, except on a processor that has locally inhibited them.

The problem is the generalisation to a multiprocessor of a familiar and well-understood uniprocessor problem. It may be reduced to a uniprocessor problem by using a global test-and-set to serialise the sleeps and wakeups, which is equivalent to synchronising through a monitor. For performance and cleanliness, however, we prefer to allow the interrupt handling and process control to be multiprocessed.

Our attempts to solve the sleep/wakeup problem in Plan 9 [Pik90] prompted this paper. We implemented solutions several times over several months and each time convinced ourselves — wrongly — they were correct. Multiprocessor algorithms can be difficult to prove correct by inspection and formal reasoning about them is impractical. We finally developed an algorithm we trust by verifying our code using an empirical testing tool. We present that code here, along with some comments about the process by which it was designed.

History

Since processes in Plan 9 and the UNIX® system have similar structure and properties, one might ask if UNIX `sleep` and `wakeup`[Bac86] could not easily be adapted from their standard uniprocessor implementation to our multiprocessor needs. The short answer is, no.

The UNIX routines take as argument a single global address that serves as a unique identifier to connect the wakeup with the appropriate process or processes. This has several inherent disadvantages. From the point of view of `sleep` and `wakeup`, it is difficult to associate a data structure with an arbitrary address; the routines are unable to maintain a state variable recording the status of the event and processes. (The reverse is of course easy — we could require the address to point to a special data structure — but we are investigating UNIX `sleep` and `wakeup`, not the code that calls them.) Also, multiple processes sleep 'on' a given address, so `wakeup` must enable them all, and let process scheduling determine which process actually benefits from the event. This is inefficient; a queueing mechanism would be preferable but, again, it is difficult to associate a queue with a general address. Moreover, the lack of state means that `sleep` and `wakeup` cannot know what the corresponding process (or interrupt) is doing; `sleep` and `wakeup` must be executed atomically. On a uniprocessor it suffices to disable interrupts during their execution. On a multiprocessor, however, most processors can inhibit interrupts only on the current processor, so while a process is executing `sleep` the desired interrupt can come and go on another processor. If the wakeup is to be issued by another process, the problem is even harder. Some inter-process mutual exclusion mechanism must be used, which, yet again, is difficult to do without a way to communicate state.

In summary, to be useful on a multiprocessor, UNIX `sleep` and `wakeup` must either be made to run atomically on a single processor (such as by using a monitor) or they need a richer model for their communication.

The design

Consider the case of an interrupt waking up a sleeping process. (The other case, a process awakening a second process, is easier because atomicity can be achieved using an interlock.) The sleeping process is waiting for some event to occur, which may be modeled by a condition coming true. The condition could be just that the event has happened, or something more subtle such as a queue draining below some low-water mark. We represent the condition by a function of one argument of type `void*`; the code supporting the device generating the interrupts provides such a function to be used by `sleep` and `wakeup` to synchronise. The function returns `false` if the event has not occurred, and `true` some time after the event has occurred. The `sleep` and `wakeup` routines must, of course, work correctly if the event occurs while the process is executing `sleep`.

We assume that a particular call to `sleep` corresponds to a particular call to `wakeup`, that is, at most one process is asleep waiting for a particular event. This can be guaranteed in the code that calls `sleep` and `wakeup` by appropriate interlocks. We also assume for the moment that there will be only one interrupt and that it may occur at any time, even before `sleep` has been called.

For performance, we desire that multiple instances of `sleep` and `wakeup` may be running simultaneously on our multiprocessor. For example, a process calling `sleep` to await a character from an input channel need not wait for another process to finish executing `sleep` to await a disk block. At a finer level, we would like a process reading from one input channel to be able to execute `sleep` in parallel with a process reading from another input channel. A standard approach to synchronisation is to interlock the channel 'driver' so that only one process may be executing in the channel code at once. This method is clearly inadequate for our purposes; we need fine-grained synchronisation, and in particular to apply interlocks at the level of individual channels rather than at the level of the channel driver.

Our approach is to use an object called a *rendezvous*, which is a data structure through which `sleep` and `wakeup` synchronise. (The similarly named construct in Ada is a control structure; ours is an unrelated data structure.) A rendezvous is allocated for each active source of events: one for each I/O channel, one for each end of a pipe, and so on. The rendezvous serves as an interlockable structure in which to record the state of the sleeping process, so that `sleep` and `wakeup` can communicate if the event happens before or while `sleep` is executing.

Our design for `sleep` is therefore a function

```
void sleep(Rendezvous *r, int (*condition)(void*), void *arg)
```

called by the sleeping process. The argument `r` connects the call to `sleep` with the call to `wakeup`, and is part of the data structure for the (say) device. The function `condition` is described above; called with argument `arg`, it is used by `sleep` to decide whether the event has occurred. `Wakeup` has a simpler specification:

```
void wakeup(Rendezvous *r).
```

`Wakeup` must be called after the condition has become true.

An implementation

The `Rendezvous` data type is defined as

```
typedef struct{
        Lock    l;
        Proc    *p;
}Rendezvous;
```

Our `Locks` are test-and-set spin locks. The routine `lock(Lock *l)` returns when the current process holds that lock; `unlock(Lock *l)` releases the lock.

Here is our implementation of `sleep`. Its details are discussed below. `Thisp` is a pointer to the current process on the current processor. (Its value differs on each processor.)

```
void
sleep(Rendezvous *r, int (*condition)(void*), void *arg)
{
        inhibit();              /* interrupts */
        lock(&r->l);

        /*
         * if condition happened, never mind
         */
        if((*condition)(arg)){
                unlock(&r->l);
                allow();        /* interrupts */
                return;
        }

        /*
         * now we are committed to
         * change state and call scheduler
         */
        if(r->p)
                error("double sleep %d %d", r->p->pid, thisp->pid);
        thisp->state = Wakeme;
        r->p = thisp;
        unlock(&r->l);
        allow(s);       /* interrupts */
        sched();        /* relinquish CPU */
}
```

Here is `wakeup`.

```
void
wakeup(Rendezvous *r)
{
        Proc *p;
        int s;

        s = inhibit();  /* interrupts; return old state */
        lock(&r->l);
        p = r->p;
        if(p){
                r->p = 0;
                if(p->state != Wakeme)
                        panic("wakeup: not Wakeme");
                ready(p);
        }
        unlock(&r->l);
        if(s)
                allow();
}
```

`Sleep` and `wakeup` both begin by disabling interrupts and then locking the rendezvous structure. Because `wakeup` may be called in an interrupt routine, the lock must be set only with interrupts disabled on the current processor, so that if the interrupt comes during `sleep` it will occur only on a different processor; if it occurred on the processor executing `sleep`, the spin lock in `wakeup` would hang forever. At the end of each routine, the lock is released and processor priority returned to its previous value. (`Wakeup` needs to inhibit interrupts in case it is being called by a process; it is a no-op if called by an interrupt.)

`Sleep` checks to see if the condition has become true, and returns if so. Otherwise the process posts its name in the rendezvous structure where `wakeup` may find it, marks its state as waiting to be awakened (this is for error checking only) and goes to sleep by calling `sched()`. The manipulation of the rendezvous structure is all done under the lock, and `wakeup` only examines it under lock, so atomicity and mutual exclusion are guaranteed.

`Wakeup` has a simpler job. When it is called, the condition has implicitly become true, so it locks the rendezvous, sees if a process is waiting, and readies it to run.

Discussion

The synchronisation technique used here is similar to known methods, even as far back as Saltzer's thesis [Sal66]. The code looks trivially correct in retrospect: all access to data structures is done under lock, and there is no place that things may get out of order. Nonetheless, it took us several iterations to arrive at the above implementation, because the things that *can* go wrong are often hard to see. We had four earlier implementations that were examined at great length and only found faulty when a new, different style of device or activity was added to the system.

Here, for example, is an incorrect implementation of wakeup, closely related to one of our versions.

```
void
wakeup(Rendezvous *r)
{
        Proc *p;
        int s;

        p = r->p;
        if(p){
                s = inhibit();
                lock(&r->l);
                r->p = 0;
                if(p->state != Wakeme)
                        panic("wakeup: not Wakeme");
                ready(p);
                unlock(&r->l);
                if(s)
                        allow();
        }
}
```

The mistake is that the reading of `r->p` may occur just as the other process calls `sleep`, so when the interrupt examines the structure it sees no one to wake up, and the sleeping process misses its wakeup. We wrote the code this way because we reasoned that the fetch `p = r->p` was inherently atomic and need not be interlocked. The bug was found by examination when a new, very fast device was added to the system and sleeps and interrupts were closely overlapped. However, it was in the system for a couple of months without causing an error.

How many errors lurk in our supposedly correct implementation above? We would like a way to guarantee correctness; formal proofs are beyond our abilities when the subtleties of interrupts and multiprocessors are involved. With that in mind, the first three authors approached the last to see if his automated tool for checking protocols [Hol91] could be used to verify our new `sleep` and `wakeup` for correctness. The code was translated into the language for that system (with, unfortunately, no way of proving that the translation is itself correct) and validated by exhaustive simulation.

The validator found a bug. Under our assumption that there is only one interrupt, the bug cannot occur, but in the more general case of multiple interrupts synchronising through the same condition function and rendezvous, the process and interrupt can enter a peculiar state. A process may return from `sleep` with the condition function false if there is a delay between the condition coming true and `wakeup` being called, with the delay occurring just as the receiving process calls `sleep`. The condition is now true, so that process returns immediately, does whatever is appropriate, and then (say) decides to call `sleep` again. This time the condition is false, so it goes to sleep. The wakeup process then finds a sleeping process, and wakes it up, but the condition is now false.

There is an easy (and verified) solution: at the end of `sleep` or after `sleep` returns, if the condition is false, execute `sleep` again. This re-execution cannot repeat; the second synchronisation is guaranteed to function under the external conditions we are supposing.

Even though the original code is completely protected by interlocks and had been examined carefully by all of us and believed correct, it still had problems. It seems to us that some exhaustive automated analysis is required of multiprocessor algorithms to guarantee their safety. Our experience has confirmed that it is almost impossible to guarantee by inspection or simple testing the correctness of a multiprocessor algorithm. Testing can demonstrate the presence of bugs but not their absence. [Dij72]

We close by claiming that the code above with the suggested modification passes all tests we have for correctness under the assumptions used in the validation. We would not, however, go so far as to claim that it is universally correct.

References

Bac86. Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs (1986).

Dij72. Edsger W. Dijkstra, ‘‘The Humble Programmer — 1972 Turing Award Lecture,’’ *Comm. ACM* **15**(10), pp. 859-866 (October 1972).

Hol91. Gerard J. Holzmann, *Design and Validation of Computer Protocols,* Prentice-Hall, Englewood Cliffs (1991).

Pik90. Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ‘‘Plan 9 from Bell Labs,’’ pp. 1-9 in *Proceedings of the Summer 1990 UKUUG Conference*,, London (July, 1990).

Sal66. Jerome H. Saltzer, *Traffic Control in a Multiplexed Computer System,* MIT, Cambridge, Mass. (1966).